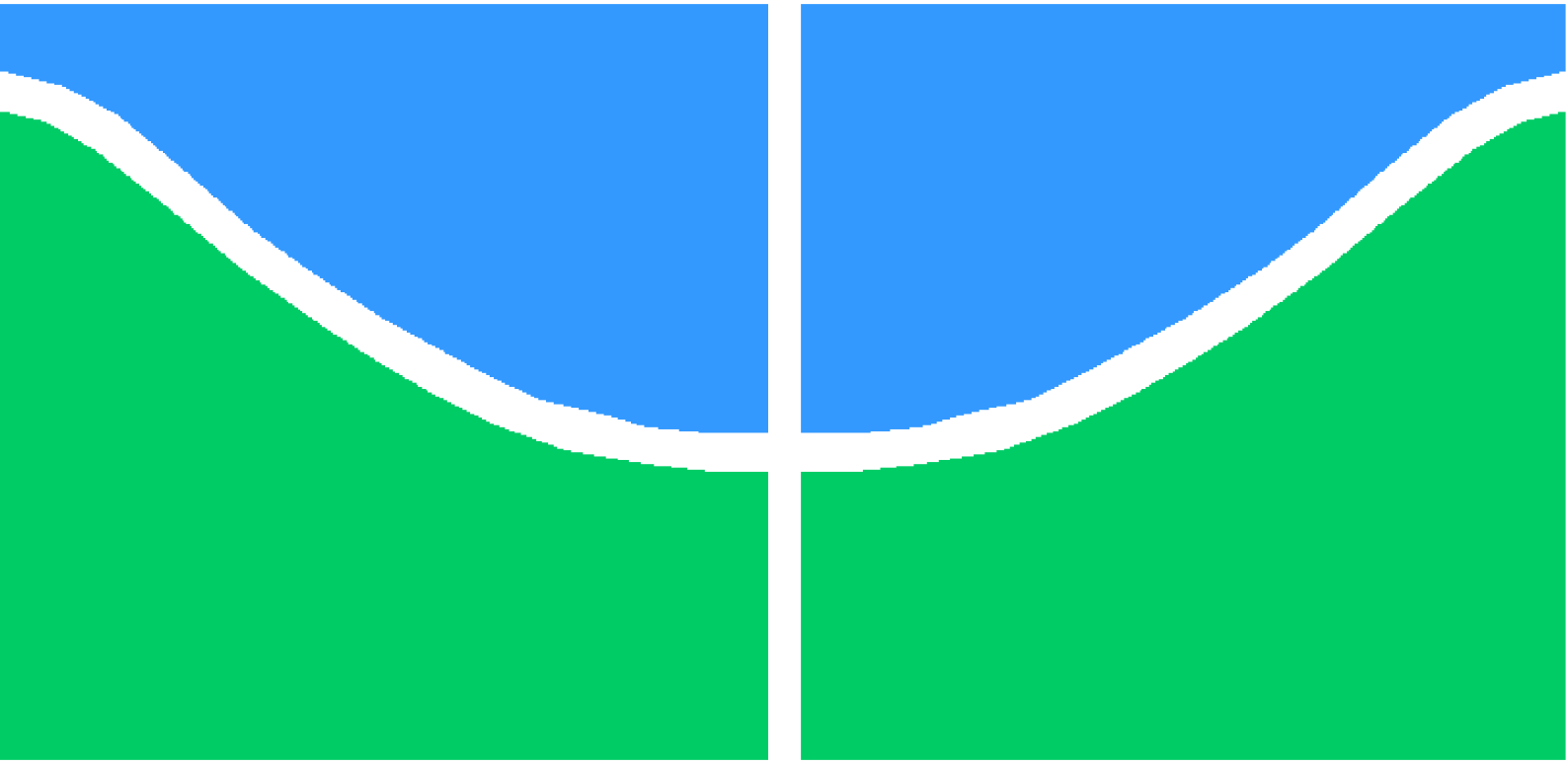

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Bacharel em Engenharia de Software

Ferramenta de apoio a auditoria de programa de segurança de software

Autor: Matheus da Silva Freire
Orientador: Prof. Dr. Fabricio Ataides Braz

Brasília, DF
2014

Matheus da Silva Freire

Ferramenta de apoio a auditoria de programa de segurança de software

Monografia submetida ao curso de graduação em Bacharel em Engenharia de Softwareda Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Bacharel em Engenharia de Software.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Fabricio Ataides Braz

Brasília, DF

2014

Matheus da Silva Freire
Ferramenta de apoio a auditoria de programa de segurança de software/ Matheus da Silva Freire. – Brasília, DF, 2014-
49 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Fabricio Ataides Braz

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2014.

1. Segurança de Software. 2. BSIMM. I. Prof. Dr. Fabricio Ataides Braz. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Ferramenta de apoio a auditoria de programa de segurança de software

CDU 02:141:005.10

Matheus da Silva Freire

Ferramenta de apoio a auditoria de programa de segurança de software

Monografia submetida ao curso de graduação em Bacharel em Engenharia de Softwareda Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 24 de junho de 2014:

Prof. Dr. Fabricio Ataides Braz
Orientador

Prof. Dr. Paulo Roberto Miranda Meirelles
Convidado 1

Prof. Dr. Luiz Augusto Fontes Laranjeira
Convidado 2

Brasília, DF
2014

Resumo

Muitos métodos foram criados para o desenvolvimento de software seguro, que vão desde a inserção de ferramentas e técnicas como a entrada pessoas especializada em construção de software seguro. Com a utilização de diferentes métodos teve iniciou o surgimento de metodologias para inserir a ideia de construção de software durante todo o desenvolvimento, a partir desta surgiu 2 (dois) modelos de maturidade que as empresas podem se basear para ter um software seguro. Porém estes modelos tem concepção diferente e através disso foi escolhido somente um, para ser objeto de referência deste trabalho. Com a adesão de um dos modelos, há a necessidade de saber o estado que a organização que o adotou possui, para identificar se seu investimento está sendo bem empregado, com intuito de ter um portfólio de software seguro.Foi criado um protótipo de ferramenta para auxiliar as organizações a entender a situação do seu programa de segurança de software, de acordo com as entidade do modelo escolhido.

Palavras-chaves: Segurança de software, maturidade, BSIMM.

Abstract

Many method have been created for the development of secure software, ranging from the insertion of tools and techniques as the input specialized people build secure software. With the use of different methdos ahd initiated the emergence of mehtodologies to insert the idea of building software through development, emerged from this two maturity models that companies can build to having a safe software. But the models have a different conception and through that we choose only one, to be the object of reference for this work. With a membership has adopted to identify wheter your investment is being well spend, in order to have a portfolio of secure software. A prototype tool to assist organizations to understand the situation of your security program software was created, according to the entities of the chosen model

Key-words: Software security, maturity, BSIMM.

Sumário

1 Introdução 17
1.1 Contextualização 17
1.2 Problema 18
1.3 Objetivo 18
1.3.1 Geral 18
1.3.2 Específico 18
1.4 Justificativa 18
1.5 Método 19
1.5.1 Etapa 1 19
1.5.2 Etapa 2 20
1.5.3 Etapa 3 20

2 Desenvolvimento 21
2.1 Etapa 1 21
2.1.1 Avaliação 23
2.2 Etapa 2 27
2.2.1 *Contexts and Dependecy Injection* (CDI) 28
2.2.2 *JavaServer Faces* (JSF) 30
2.2.3 Hibernate 32
2.2.4 Arquillian 32
2.3 Etapa 3 33
2.3.1 Testes 40
2.4 Exemplo de Uso 42

3 Considerações 45
3.1 Trabalhos Futuros 46

Referências 47

Lista de ilustrações

Figura 1 – Domínios e Práticas do BSSIM. McGraw, Migues e West (2013) 22
Figura 2 – *Scorecard* do BSIMM com as atividades levantadas. (MCGRAW; MIGUES; WEST, 2013) 24
Figura 3 – Esboço Inicial da Arquitetura 29
Figura 4 – Visão geral do ciclo de vida do JSF(CORDEIRO, 2013) 31
Figura 5 – Diagrama de pacotes do sistema 34
Figura 6 – Modelo de Dados do sistema 35
Figura 7 – Diagrama de sequência do sistema 37
Figura 8 – Página de login 38
Figura 9 – Página de visualização das auditorias do perfil logado 38
Figura 10 – Tela de seleção das empresas vinculadas ao perfil 39
Figura 11 – Página dos resultado da busca de auditorias pela data 39
Figura 12 – Página para criar uma nova auditoria 40
Figura 13 – Diálogo que aparece caso uma opção de auditoria exista, com os mesmo parâmetros de uma nova 40
Figura 14 – Página de uma nova auditoria 41
Figura 15 – Diálogo para inserir uma nova resposta 41
Figura 16 – Gráfico de teia sobre a auditoria somente em uma prática 42
Figura 17 – Gráfico de teia sobre a auditoria 42
Figura 18 – Auditoria em um segundo momento 43
Figura 19 – Terceira auditoria 44

Lista de tabelas

Tabela 1 – Palavras-Chave 20

Tabela 2 – Requisitos funcionais da ferramenta 28

Lista de abreviaturas e siglas

BSIMM	*Building Security In Maturity Model*
CDI	*Contexts and Dependecy Injection*
JSF	*JavaServer Faces*
JSP	*JavaServer Pages*
MVC	*Model-View-Controller*
OO	Orientação a objetos
OWASP	*Open Web Application Security Project*
PT	*Penetration Testing*
SAMM	*Software Assurance Maturity Model*
SDL	*Software Development Lifecycle*
SDLC	*Software Development Life Cycle*
SSG	*Software Security Group*
XML	*eXtensible Markup Language*

1 Introdução

A segurança da informação e, em especial, o aspecto da privacidade de dados, tem sido assunto em destaque nos meios de comunicação. O noticiário tem relatado com frequência ocorrências negativas relacionadas a esse tema, revelando uma constante relativização da ética por todos os países/organizações com capacidade técnica para executar ataques, espionagem, fraudes e outras ações intrusivas que comprometam o sigilo, a integridade ou disponibilidade da informação.

Na tentativa de impor maior dificuldade aos adversários em conseguir alcançar os seus objetivos escusos, a segurança da informação como um todo necessita ser reforçada. Uma parte desse todo é a segurança de software, tema com o qual este trabalho de conclusão de curso procurou contribuir.

1.1 Contextualização

A segurança da informação é um campo repleto de desafios. Um grande desafio para os defensores, pessoas com a função de garantir as propriedades de segurança de seu sistema, é o fato dos atacantes necessitarem de apenas uma brecha ou vulnerabilidade para comprometer o sistema. Por mais que o defensor eleve o nível de segurança de seu sistema, se uma única falha for deixada de lado, ela poderá ser a porta de entrada usada pelo atacante para abusar do sistema.

Esse desafio se torna ainda maior, quando se compara o recurso disponível para uma organização se defender, face ao efetivo de atacantes focado na exploração de seu sistema. Tais adversários podem ser motivados por recompensas intangíveis, como, por exemplo, a notoriedade pela execução do ataque; pelo retorno financeiro decorrente da comercialização da informação como dados de cartões de crédito; ou pela vantagem comercial em função da revelação de estratégias de negócio de uma empresa/país. Toda essa adversidade impõe à organização a obrigação de lidar estrategicamente com os recursos disponíveis para a segurança da informação. Desta forma, é imperativo que aconteça um fortalecimento dos elos que compõe a segurança da informação como a segurança de rede, a segurança de estação (*host*) e segurança de software.

O elo da segurança de software demorou a receber atenção, quando se comparado com os outros mencionados. Enquanto as iniciativas em antivírus e filtro de pacotes estão em evidência há mais de 30 anos, a segurança de software começou a receber maior atenção a partir do ano 2000, com a ampla divulgação do programa *Trustworthy Computing Initiative* da *Microsoft*, pelo qual a empresa elevou a segurança de software como a sua

prioridade estratégica (VIEGA, 2011).

1.2 Problema

A dificuldade das organizações em conduzir de maneira profissional um programa de segurança de software, em especial evidenciar o estado, em termos de segurança de software, em que a organização se encontra.

1.3 Objetivo

Esta seção do trabalho apresenta o objetivo que se deseja conquistar, e para alcançar tal objetivo houve a divisão em etapas, para facilitar o desenvolvimento do trabalho, essas etapas são chamadas objetivos específicos.

1.3.1 Geral

O objetivo principal deste trabalho é facilitar o processo de levantamento do estado em termos de segurança de software em que a organização se encontra.

1.3.2 Específico

- Escolher um modelo de segurança base para o diagnóstico;
- Prototipar ferramenta que suporta a análise do estado de segurança;

  Projetar a arquitetura de software para a ferramenta

  Codificar a ferramenta

1.4 Justificativa

Apesar do investimento isolado em controles/mecanismos de segurança incorrer na elevação da estado de segurança de uma determinada aplicação, isso não resulta no amadurecimento da postura de segurança da organização como um todo. Ou seja, esse investimento não necessariamente trará ganhos de segurança para todos os aplicativos desenvolvidos, sejam novos projetos ou manutenções evolutivos, pela organização uma vez que para isso é necessário que seja elevado, não somente a capacidade em segurança de determinada equipe, mas a cultura em segurança da organização.

Segundo NIST 82% das vulnerabilidades estão no software e é necessário pensar em segurança nas demais fases do ciclo de desenvolvimento do que somente a codificação. Segundo Chess e Arkin (2011) existem algumas atividades que devem ser empregadas

durante todo o desenvolvimento, entretanto, essas atividades devem ser encarada como um início de um programa para que a organização possa ter proveito das lições aprendidas na sua cultura organizacional(equipe com conhecimento e ferramentas especializadas), entre tais atividades podemos citar:

- Manter uma equipe de segurança: uma equipe que ficará responsável pela segurança de software de toda organização;
- Manter códigos legados: códigos antigos podem ser um guia para organização de como fazer correto, como também um exemplo de como não ser feito;
- Estabelecer um programa de treinamento e educação: este programa irá auxiliar a organização como um todo em obter segurança e aprender durante todo o desenvolvimento;
- Estabelecer padrões e métricas: mantendo um padrão de codificação, de levantamento de requisito, a organização pode ter uma nível de segurança alvo, e assim auxiliar a equipe de métricas a determinar qual o estado da sua organização.
- Manter um canal de *feedback* para melhoria contínua: esse canal pode ser interno ou ter lições aprendidas com erros em aplicativos para que não se repitam.

A partir dessas atividades é necessário que a empresa tenha um grau de maturidade acentuado para determinar que somente áreas específicas não irão aumentar o nível de segurança de uma aplicação, e então o uso de um modelo de maturidade deve ser estudado e se for decidido pela adoção, deve haver uma avaliação com base no modelo.

A auditoria de segurança de software de uma organização/programa de segurança visa auxiliar a organização a saber quais pontos devem ser melhorados, ou quais pontos ao longo de tempo sofreram alteração prejudicial ou não.

1.5 Método

Para que o trabalho possa ser executado e posteriormente reproduzido, foi definido um método organizado em etapas. Essas etapas estão descritas nas sub-seções seguintes.

1.5.1 Etapa 1

A etapa 1 consiste em determinar qual o modelo de segurança a ser seguido para o desenvolvimento da ferramenta. Para a escolha de quais modelos possam ser utilizados, levou-se em conta a natureza dos modelos disponíveis, destaca-se a natureza de modelos de maturidade focado na organização como um todo e foi excluído modelos que tenham a visão em apenas pontos do desenvolvimento.

Para a pesquisa bibliográfica deste trabalho utilizou-se sites sobre artigos científicos, como por exemplo o *Google Scholar* onde as palavras chaves que guiaram a pesquisa encontra-se na Tab.(1).

Tabela 1 – Palavras-Chave

Paralavras Chave
Building Security In Maturity Model (BSIMM)
open*Software Assurance Maturity Model* (SAMM)
Security model for software
Software security
BSIMM and openSAMM models

Outros meios de pesquisa foram:

1. IEEE - *Institute of Electrical and Electronics Engineers* ou Instituto de Engenheiros Eletricistas e Eletrônicos
2. ACM - *Association for Computing Machinery*

1.5.2 Etapa 2

A etapa 2 consiste na definição das decisões arquiteturais para a implementação da ferramenta. As principais decisões tomadas durantes esta etapa são a de frameworks utilizados, linguagem, ferramentas para o desenvolvimento e o levantamento dos requisitos.

1.5.3 Etapa 3

Esta última etapa consiste na implementação da ferramenta. Nesta etapa também esta contemplado o desenvolvimento de testes a fim de verificar que a ferramenta, ou protótipo, está atendendo os requisitos levantados.

2 Desenvolvimento

2.1 Etapa 1

Em um mercado em que a concorrência mostra-se acirrada a probabilidade de encontrar um produto de software com qualidade duvidosa é grande. Para que a aquisição/utilização seja considerada, o software deve ter sua qualidade atestada por padrões de confiança para ter uma solução eficaz e duradoura e, ao mesmo tempo, substituível. Para atingir esse patamar, as organizações devem desenvolver e manter a competência adequada - com maior eficiência e eficácia; o exercício contínuo de melhoria faz com a organização atinja um patamar elevado de maturidade de acordo com as suas possibilidades e objetivos empresariais (CHRISSIS; KONRAD; SHRUM, 2003).

Maturidade é um objetivo que a empresa almeja, que depende da tecnologia, metodologia e gestão da época (TONINI; CARVALHO; SPINOLA, 2008). Ter um nível de maturidade, um estado, é a competência de identificar e buscar o nível necessário e suficiente, através da obtenção de conhecimento, de desenvolvimento de habilidades e capacidade de adaptação para com os objetivos propostos. Essa maturidade em software pode ser obtida através da adoção de modelos de maturidade, que fornece às organizações de software um guia de como obter controle em seus processos para desenvolver e manter software e como evoluir em direção a uma cultura de engenharia de software e excelência em desenvolvimento. A qualidade de software tem aumentado com a adoção de modelos de maturidade durante o desenvolvimento, porém os modelos não devem ser percebidos como a solução final para acabar com a baixa qualidade que existe em alguns software, a adoção deve ser encarada como um dos fatores a mais para que a qualidade tenha aumento (REGULWAR; GULHANE; JAWANDHIYA, 2010).

Os modelos de maturidade, o CMMI (*Capability Maturity Model Integration*) e o MPS.Br (Melhoria do Processo de Software Brasileiro), são pautadas na ideia de prescrição de atividades e técnicas para inserção dentro do processo de desenvolvimento. Na segurança de software também há modelos de maturidade que auxiliam as organizações a desenvolver e manter um software seguro, um modelo de maturidade enumera práticas, técnicas e atividades para o a construção de software que permaneça seguro face a um ataque, erro.

A ideia, de construção e manutenção de software seguro, dentro da organização começa nos anos 2000 com desenvolvimento de alguns processos como, por exemplo o *Software Development Lifecycle* (SDL) da *Microsoft*(MCGRAW, 2012);além de modelos para profissionalizar um programa de segurança como, por exemplo, OpenSAMM (SAMM

da *Open Web Application Security Project* (OWASP) e o BSIMM da Cigital. O objetivo destes modelos é estabelecer uma medida da maturidade de segurança, além de ser um guia para o a progresso da cultura da organização em relação a segurança de software.

Apesar dos modelos OpenSAMM e BSIMM serem pautados na melhoria da maturidade da organização, a natureza deles é distinta, uma vez que o primeiro se coloca como um modelo prescritivo, ou seja, ele enumera ações que, na opinião de alguns especialistas, devem ser empreendidas para uma organização produzir software seguro. O segundo se propõe a ser um modelo descritivo, ou seja, nele encontram-se consolidadas observações realizadas em campo. Essa diferenciação confere ao modelo BSIMM um caráter científico ausente no OpenSAMM, fato que justifica a preferência por ele no desenvolvimento deste trabalho de conclusão de curso.

Segundo McGraw, Migues e West (2013), o modelo deve ser encarado como um modelo de maturidade porquê, geralmente, a mudança, em introduzir segurança dentro do ciclo de desenvolvimento, é uma mudança na filosofia de trabalho da organização, e isto não acontece da noite para o dia.

O BSIMM considera preocupações do interesse de um programa de segurança de software. Essas preocupações estão representadas em 109 atividades, distribuídas em 12 práticas, classificadas nos quatro domínios ilustrados pela Figura 1.

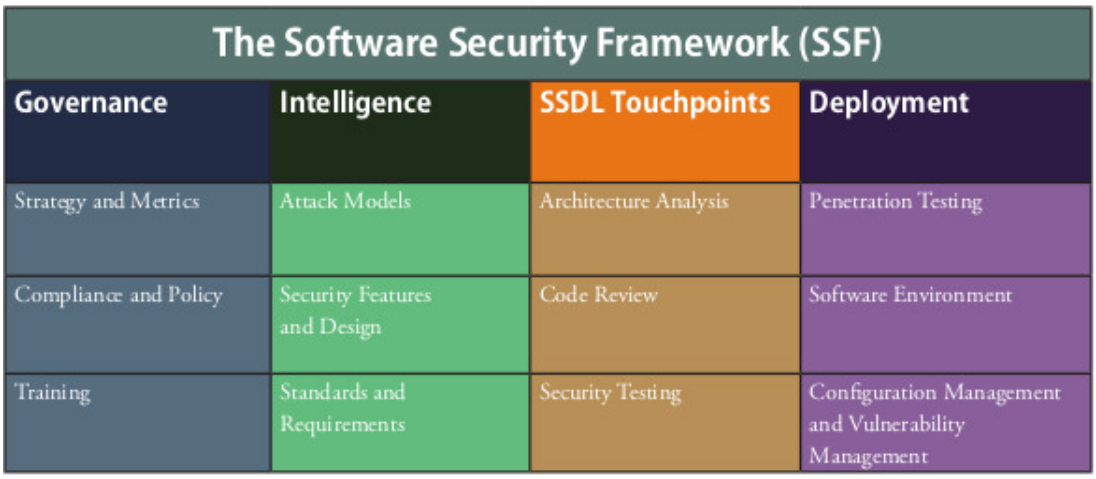

Figura 1 – Domínios e Práticas do BSSIM. McGraw, Migues e West (2013)

Mais detalhes sobre o domínio são dados a seguir, no qual uma breve explanação do significado de cada um é apresentada:

- Governança: são práticas e métricas que envolvem planejamento, atribuição de papéis e responsabilidades. Identifica os objetivos, nível alvo, além do custo a ser gasto durante todo o programa ;

- Inteligência: o domínio visa identificar recursos utilizados no programa de segurança de software da organização;

- SSDL (*software security development lifecycle*) Touchpoints: práticas associadas com o desenvolvimento da aplicação na fase de desenvolvimento;

- Implantação: O domínio de implantação envolve práticas de verificação da aplicação em relação à segurança e verificação dos ativos que o programa entregou. (MCGRAW; MIGUES; WEST, 2013)

O BSIMM é resultado de pesquisa que observou vários programas de segurança de software com o intuito de reconhecer e consolidar atividades comuns executadas por diversas organizações. As atividades reconhecidas são categorizadas em três níveis, conforme a sua ocorrência nas organizações, ou seja, se uma atividade foi mais observada nas organizações participantes, ela é pautada como uma atividade básica, ou nível 1, à medida que a quantidade de organizações executam alguma atividade, ela vai recebendo um nível maior. Por exemplo, a atividade 1 foi vista em todas as organizações, por isso seu nível é o 1, entretanto, a atividade 2 foi percebida somente em 40% das organizações, por isso seu nível é o 2. E a atividade 3 foi observada em 10% dos participantes, por isso a atividade é categorizada como uma atividade de alto nível, ou o maior nível que o modelo informa, que é o 3.

A Figura 2 apresenta uma visão consolidada da ocorrência de cada uma das práticas nas 67[1] organizações alvo da pesquisa, que envolve participantes dos mais variados ramos (empresas de energia, telefonia, saúde, setor financeiro, etc.).

A Figura 2, o modelo dá a visão de como a atividade foi observada na pesquisa do BSIMM V. Por exemplo, a prática de *Penetration Testing* (PT) é dividida em 3 níveis, contendo certas atividades por níveis. A atividade 1, PT1.1, foi observada em 62 das 67 organizações participantes.

A escolha do BSIMM deu-se pela grande quantidade de casos relatados da sua adoção, e outra característica que o deixa como a referência principal deste trabalho é que, periodicamente, retira da sua base de dados, informações que contenham mais de 48 meses, isso é, os dados contidos no BSIMM são extremamente atuais e condizentes com a perspectiva de ser um *framework*[2] com as melhores práticas de segurança (MCGRAW; MIGUES; WEST, 2013).

[1] Estes participantes estão disponíveis em: http://bsimm.com/community/

[2] Um *framework* em software pode ser entendido como uma abstração onde as práticas ou códigos comuns provê funcionalidades genéricas e estas podem ser selecionáveis, um *framework* não pode ser alterado em sua essência (código fonte), porém se for necessário, pode ser modificado para atender a necessidade vigente.

Governance		Intelligence		SSDL Touchpoints		Deployment	
Activity	Observed	Activity	Observed	Activity	Observed	Activity	Observed
[SM1.1]	44	[AM1.1]	21	[AA1.1]	56	[PT1.1]	62
[SM1.2]	34	[AM1.2]	43	[AA1.2]	35	[PT1.2]	51
[SM1.3]	34	[AM1.3]	30	[AA1.3]	24	[PT1.3]	43
[SM1.4]	57	[AM1.4]	12	[AA1.4]	42	[PT2.2]	24
[SM1.6]	36	[AM1.5]	42	[AA2.1]	10	[PT2.3]	27
[SM2.1]	26	[AM1.6]	16	[AA2.2]	8	[PT3.1]	13
[SM2.2]	31	[AM2.1]	7	[AA2.3]	20	[PT3.2]	8
[SM2.3]	27	[AM2.2]	11	[AA3.1]	11		
[SM2.5]	20	[AM3.1]	4	[AA3.2]	4		
[SM3.1]	16	[AM3.2]	6				
[SM3.2]	6						
[CP1.1]	43	[SFD1.1]	54	[CR1.1]	24	[SE1.1]	34
[CP1.2]	52	[SFD1.2]	53	[CR1.2]	34	[SE1.2]	61
[CP1.3]	45	[SFD2.1]	26	[CR1.4]	50	[SE2.2]	31
[CP2.1]	24	[SFD2.2]	29	[CR1.5]	23	[SE2.4]	25
[CP2.2]	28	[SFD2.3]	9	[CR1.6]	25	[SE3.2]	10
[CP2.3]	29	[SFD3.1]	13	[CR2.2]	10	[SE3.3]	9
[CP2.4]	25	[SFD3.2]	9	[CR2.5]	15		
[CP2.5]	35			[CR3.1]	18		
[CP3.1]	14			[CR3.2]	4		
[CP3.2]	11			[CR3.3]	6		
[CP3.3]	8			[CR3.4]	1		
[T1.1]	50	[SR1.1]	48	[ST1.1]	51	[CMVM1.1]	59
[T1.5]	29	[SR1.2]	43	[ST1.3]	55	[CMVM1.2]	59
[T1.6]	23	[SR1.3]	45	[ST2.1]	27	[CMVM2.1]	50
[T1.7]	33	[SR1.4]	27	[ST2.3]	13	[CMVM2.2]	44
[T2.5]	9	[SR2.1]	23	[ST2.4]	11	[CMVM2.3]	30
[T2.6]	13	[SR2.2]	19	[ST3.1]	8	[CMVM3.1]	6
[T2.7]	9	[SR2.3]	19	[ST3.2]	6	[CMVM3.2]	6
[T3.1]	4	[SR2.4]	22	[ST3.3]	5	[CMVM3.3]	2
[T3.2]	4	[SR2.5]	8	[ST3.4]	7		
[T3.3]	8	[SR3.1]	12				
[T3.4]	9						
[T3.5]	5						

Figura 2 – *Scorecard* do BSIMM com as atividades levantadas. (MCGRAW; MIGUES; WEST, 2013)

2.1.1 Avaliação

Com a adoção de um modelo para melhorar o processo, envolve a identificação de problemas e oportunidades de um aperfeiçoamento ainda na fase inicial. A identificação

dos problemas que a organização possui e as melhorias que efetivamente possam trazer algum benefício no desenvolvimento de software é essencial para o sucesso da adoção do modelo. Para que a definição dos problemas mais prioritários, assim como as melhores oportunidades que incremente os resultados da adoção de modelos de maturidade, seja bem realizada, as empresas, atualmente, estão recorrendo mais vezes a avaliação dos processos (ANACLETO; WANGENHEIM, 2004). Esta avaliação geralmente ocorre com o objetivo de obter conhecimento sobre a situação do estado do processo pela organização. A avaliação de processos consiste basicamente em uma medição de aspectos relevantes às metas de melhoria (BASILI; CALDIERA; ROMBACH, 1994).

A execução de uma avaliação também permite uma comparação entre os processos como são executados antes e após o programa de melhoria. Dessa forma é viabilizada uma avaliação do desempenho dos processos e dos resultados obtidos com a realização das ações de melhoria definidas no programa.

Cada modelo prevê seu próprio método de avaliação, as ISO 15504 e 9000 apresentam etapas mínimas para que uma avaliação seja realizada, exceto nos casos que seguem a ISO 9000, estes são chamados de auditoria, não existindo grande diferença entre auditoria e avaliação. Em alguns casos o que considera-se como diferença da auditoria é a execução ser realizada por terceiros e tem por resultado, por exemplo, um relatório. Ambas podem ser de duas formas, interna e externa, a interna é executada pela própria empresa, a externa é o processo de análise do processo (ferramentas, técnicas e pessoas) da contrante (ANACLETO; WANGENHEIM, 2004).

Segundo Anacleto e WANGENHEIM (2004) é comum as avaliações de modelos ter fases, ou etapas, similares e são essas:

1. Planejamento: planejamento das atividades a serem executadas, preparação da equipe que irá atuar, definição de metas, cronogramas e demais atividades de planejamento;

2. Coleta e Validação de evidências: realização de entrevistas, aplicação de questionários, análise de documentos e demais atividades que permitam verificar como os processos são executadas e que produtos são gerados/utilizados, permitindo assim uma coleta de evidência da execução destes processos.

3. Verificação do atendimento aos requisitos da norma/modelo: com base nas evidências coletadas é feita uma verificação se os requisitos definidos no modelo para certificação ou para determinação do nível de capacidade do processo são atendidos. Assim como são identificados pontos fortes e fracos dos processo; e

4. Registro dos resultados: os resultados finais são então apresentados para a equipe e armazenados em um relatório conforme definido no modelo.

As ISO apresentam apenas requisitos mínimos de um processo para que a avaliação seja conforme o modelo, como também não indica como deve acontecer. Um exemplo do método de avaliação de um modelo é o SCAMPI, do modelo CMMI, onde é descrito atividades obrigatórias e pontos que podem ser adaptados.

A base de uma avaliação SCAMPI está na verificação dos indicadores das práticas implementadas que o CMMI indica que devem ser realizadas, essa verificação tem como base, as evidências produzidas pela execução da prática (ITABORAHY et al., 2005). Uma avaliação tem o objetivo de levantar a situação da implantação dos modelos, identificar pontos fortes e pontos fracos (oportunidades de melhoria) na área que foi alvo da avaliação, servir de base para identificar áreas do processo de desenvolvimento que necessitem de um maior investimento (PRIKLADNICKI; BECKER; YAMAGUTI, 2005). Além de realizar uma avaliação, as empresas realizam um diagnóstico de seus processos para verificar se está ao nível pretendido. Tendo como objetivo identificar a situação da empresa antes de uma avaliação formal, sem o rigor de uma avaliação.

O método SCAMPI possui diferentes classes de avaliação, que são as classes A, B e C. A classe A é a avaliação formal, onde o resultado desta é a atestação do nível de maturidade da empresa, a classe B é considerada uma avaliação não formal, onde se identifica possíveis pontos de melhorias para quando a avaliação for executada, ser bem sucedida. Já a classe C é a identificação de pontos que, de acordo com o objetivo da empresa, necessitem de melhorias, porém todas estas classes de avaliação seguem o mesmo fluxo de desenvolvimento. A avaliação feita com o SCAMPI, segue o fluxo de quatros fases que são: Preparação e planejamento, Condução da avaliação, Resultados e Certificação, esse é um fluxo bastante linear onde é necessário que uma etapa esteja concluída para que uma possa começar.

Tendo como referência os modelos de maturidade de segurança, estes não possuem um documento específico para a sua avaliação, entretanto, como ambos são modelos de maturidade que aferem nível de uma organização, não é errôneo falar que estes podem utilizar as ISOs para tal atividade.

Para avaliação do modelo escolhido, o BSIMM informa que a melhora forma é a comparação com os dados que o modelo apresenta, isso é, são dados divididos em todos os participantes, dividido em áreas de negócio etc. Com isso, o objetivo de avaliação deve ser escolhido de acordo com as práticas e podendo ser acrescido de atividades que façam sentido. A avaliação da maturidade utilizando o BSIMM tem o objetivo de observar evolução, mudanças e melhorias ao longo do tempo.

A avaliação utilizando o modelo, mostra que a sua utilização em organizações de software é bastante possível e extremamente útil. A utilização do BSIMM pode ser usado para planejar, estruturar um novo programa de segurança de software e a sua execução pode ajudar a evoluir a iniciativa da segurança de software dentro da organização. Segundo

o modelo, McGraw, Migues e West (2013), é notório que as organizações participantes tiveram melhorias em seus programas de segurança com a sua adoção.

2.2 Etapa 2

Conforme falado no 1.5.2, nesta etapa é a definição da arquitetura do sistema. A arquitetura de software é a representação das decisões tomadas para o desenvolvimento de software, essas decisões devem satisfazer os requisitos levantados (PRESSMAN, 2006).

A arquitetura de software deve ser entendida como uma definição das relação/interações entre o ambiente e os componentes do sistema (WILLIAMS; CARVER, 2010 apud GUSTAFSSON et al., 2002, p. 2). Segundo Bass, Clements e Kazman (2003), a arquitetura de software tem sua importância para:

- Facilitar a comunicação entre todas as partes interessadas no desenvolvimento do sistema;
- Destacar decisões iniciais de projeto que terão impacto profundo em todo o trabalho da engenharia de software durante o desenvolvimento do produto;
- Constituir um modelo relativamente pequeno, inteligível de como o sistema esta estruturado e como seus componentes se interagem

Antes de tomar a decisão inicial sobre a parte lógica do ferramenta (linguagem,*frameworks*, etc.) foi necessário levantar os requisitos que deveriam ser atendidos. Estes requisitos foram levantados com base na experiência anterior do professor orientador em relação a avaliação que uma organização sofre quando se adota o modelo com isso, os requisitos levantados para a ferramenta se encontram na Tab.(2).

A decisão inicial da arquitetura, foi o tipo de sistema a ser implementado, se seria um sistema *standalone*[3] ou sistema *WEB*[4]. A decisão tomada foi de um sistema *WEB* principalmente influenciado pela escolha da linguagem de implementação, que foi a linguagem Java para aplicação *WEB*, mais conhecido como Java *WEB*.

A arquitetura escolhida é uma arquitetura que favorece a comunicação entre os diversos componentes do sistema (classes) e a coesão quanto a responsabilidade de cada classe. Dentro das decisões da arquitetura foi identificado quais as ferramentas seriam usadas durante o desenvolvimento, compõem ferramentas do sistema todos recursos utilizados para o desenvolvimento, desde a banco de dados até *frameworks* escolhidos para o desenvolvimento. O banco de dados utilizado foi postgres[5], a utilização CDI, a utilização

[3] Sistema que roda localmente na máquina do cliente
[4] Aplicação que é executada em um servidor e o cliente acessa via internet
[5] Banco de dados livre disponível para *download* no site: http://www.postgresql.org/

Tabela 2 – Requisitos funcionais da ferramenta

Requisitos	**Descrição**
O sistema deve manter o cadastro de usuários	O cadastro de usuário deverá ser efetuado para este ter a possibilidade de acessar o sistema.
O sistema deve manter o cadastro de companhias e o vínculo com os usuários	As companhias cadastradas deverão aparecer de acordo com o usuário logado, e esta companhia poderá sofrer uma auditoria.
O sistema deve manter o cadastro de versões do modelo	As versões do modelo deverá ser cadastrado para o usuário identificar qual o modelo está sendo usado na auditoria.
O sistema deve manter o cadastro de atividades do modelo e o relacionamento com as versões do modelo	O cadastro de atividade deverá ser efetuado afim de saber quais atividades a companhia está atendendo e qual seu nível. As atividades também tem relacionamento com diferentes versões do modelo.
O sistema deverá permitir o cadastro de questões	As questões deverão ser vinculadas com as atividades, independente da quantidade, e estas questões que serão exibidas para o usuário quando este realizar a auditoria.
O sistema deve permitir o cadastro de uma auditoria, podendo este ser criado dinamicamente	Uma auditoria irá revelar como se encontra o estado da organização, de acordo com os filtros selecionados pelo usuário na criação de uma nova auditoria, vale ressaltar que se se existir uma auditoria igual àquela que está sendo criada, o sistema emitirá uma mensagem para informar o usuário e saber se ele deseja utilizar as questões já vinculadas.
O sistema deve exibir gráficos da auditoria realizada.	O gráfico da auditoria será na forma de gráfico de teia, este gráfico deverá ser usado para futuras comparações do estado da companhia com as empresas que já participam da pesquisa.

para a camada de apresentação o JSF e o servidor de aplicação JBoss, além do uso de um JPA (Java Persistence API) que padroniza a *framework* ORM, *Object-Relational Mapping* e um componente para teste.

A Figura (3) mostra um esboço inicial da arquitetura, isso é, é a visão de como cada parte do sistema irá se comunicar com outras.

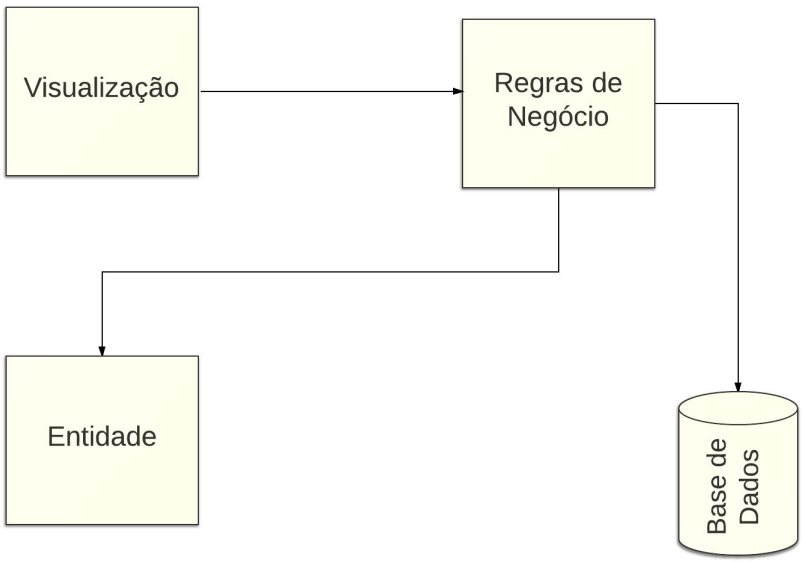

Figura 3 – Esboço Inicial da Arquitetura

2.2.1 CDI

Segundo Razina et al. (2007), a injeção de dependência é o padrão de que permite o programador a "injetar"um objeto em uma classe usando um "recipiente"(*container*) configurado externamente (geralmente um arquivo *eXtensible Markup Language* (XML)) em vez de deixar essa responsabilidade para a classe.

O CDI for JAVA EE é a especificação que rege como funciona os diversos *frameworks* deste, através dessa especificação que o CDI segue, é possível alterar o *framework* de desenvolvimento e mesmo assim a sua arquitetura não sofrer alteração (CORDEIRO, 2013). A especificação que rege o CDI no Java é a JSR 299 que define de forma unificada e contextualizada o modelo de ciclo de vida para o Java EE 6.

A injeção de dependência pode ser chamada de inversão de controle, e usualmente é tratada como: "Não me chame, eu te chamo". Esse princípio trata de um componente executar sua funções e chamar componentes que ele precisar quando for executar. A utilização do CDI resulta em um código mais robusto, menos acoplado, mais fácil de ser mantido, além de ser mais facilmente configurado em um servidor e mais facilmente testado. (BORANBAYEV, 2009)

De acordo com o JSR299, o CDI provê:

- Integração com expressões regulares (EL), que permite qualquer componente ser utilizado diretamente em um página JSF Ou *JavaServer Pages* (JSP);
- A habilidade de injetar componentes, de acordo com necessidade do uso;
- A habilidade de utilizar *interceptors*;
- Um modelo de notificação de eventos;
- Um modelo de pedido-requisição em adição com 3 (três) escopos padronizados (*request*, *session* e *application*) definido na especificação de Servlet;
- Uma provedora de interface de serviços que permite a *frameworks* distintos se integrarem com o Java.

Neste trabalho o uso do CDI foi implementado com a utilização do *framework* JBoss SEAM juntamente com o *container* WELD. O *Weld* permite a execução da aplicação em servidores como o Tomcat, Jetty ou JBoss. Sua escolha de uso foi pela experiência em desenvolvimento com esta ferramenta e o *framework* trabalhar em conjunto com diversas outras ferramentas, que é o caso de servidores de aplicações (JBoss, Glassfish, Tomcat). Além disso, o Seam provê recursos de internacionalização, segurança, integração com o JSF.

2.2.2 JSF

O JSF é um *framework* de apresentação que fornece ao programador a capacidade de criar páginas dinâmicas através de componentes (KING, 2009). O JSF simplifica o desenvolvimento de telas para apresentação ao cliente, especialmente que em sua essência, desde a versão inicial do *framework*, possui uma definição madura de processamento da página. Através dessa definição, os programadores que desejam utilizar o JSF, podem usufruir das premissas que o JSF tem em sua especificação (BERGSTEN, 2004):

- Integração facilitada com o *backend*: os programadores não precisam se preocupar como o objeto será passado para as camadas inferiores;
- Desenvolvimento de telas sem o conhecimento aprofundado em interface com o usuário: o desenvolvimento de telas não será a prioridade, porém não será um empecilho para quem deseja utilizar o JSF.

A construção de páginas com o JSF segue uma diferença notável com outros *frameworks* para aplicações em web, o JSF é baseado em componentes, ou seja, ele mantém

uma estrutura por trás da tela, e apresenta apenas o código HTML, porém o desenvolvedor não precisa se preocupar com a conversão de uma página JSF, o XHTML, para HTML, isso é responsabilidade do *framework*.

Por isso seus componentes tem um ciclo de vida a ser seguido, e com isso é necessário um entendimento sobre as etapas do seu ciclo de vida. Todo o ciclo de vida do JSF é composto por 6 fases e são essas as etapas (CORDEIRO, 2013), e na Fig. 4 é apresentado uma visão geral do JSF:

Fase 1 - *Restore View*: nesta fase do ciclo de vida é construído a estrutura de componentes da tela ou restaurar a árvore dos componentes correspondentes aos arquivos xhmtl que foi chamado. Se o JSF conseguir criar a árvore (não apresentar algum erro na estrutura interna do arquivo chamado) ele passa para a fase 2;

Fase 2 - *Apply Request Value*: Nesta fase, dentro da página chamada, o JSF coloca os atributos chamados nos componentes correspondentes, essa inserção independe de o valor ser válida ou não;

Fase 3 - *Validate*: Nesta fase o *framework* se preocupa em 2 coisas, converter o valor para o tipo de classe propício (tipo inteiro, ponto flutuante, tipo de data, etc.) e validar se este está de acordo com o atributo informado na *Bean*;

Fase 4 - *Update Model*: Nesta fase o JSF fica a cargo de atualizar a classe responsável, ou a classe *bean* que interage com a tela. Os valores dos atributos são inseridos dentro do modelo em questão.

Fase 5 - *Invoke Application*: Nesta fase a aplicação executa os comandos pré-estabelecidos, ou seja, nesta fase que o sistema se preocupa com a lógica determinada.

Fase 6 - *Render Response*: Nesta fase é a interação com o usuário final, ela é a fase responsável por entregar uma mensagem de sucesso, se todas as fases foram respeitadas, ou uma mensagem de erro se alguma das fases não foi respeitadas.

O JSF permite a flexibilidade de integração com outros *frameworks*, como por exemplo o JSP, onde o JSF faz o papel de camada mais avançada na apresentação. Além disso, a utilização do JSF permite o programador mais acostumado com a linguagem Java, a desenvolver uma aplicação com código bastantes similares ao desenvolvimento do Java swing, ou Java *standalone*.

Os maiores exemplos de *frameworks* de JSF são o *RichFaces* e o *Primefaces*, ambos se integram com as especificações do JSF 2.X, e devido a experiência com o *primefaces* este

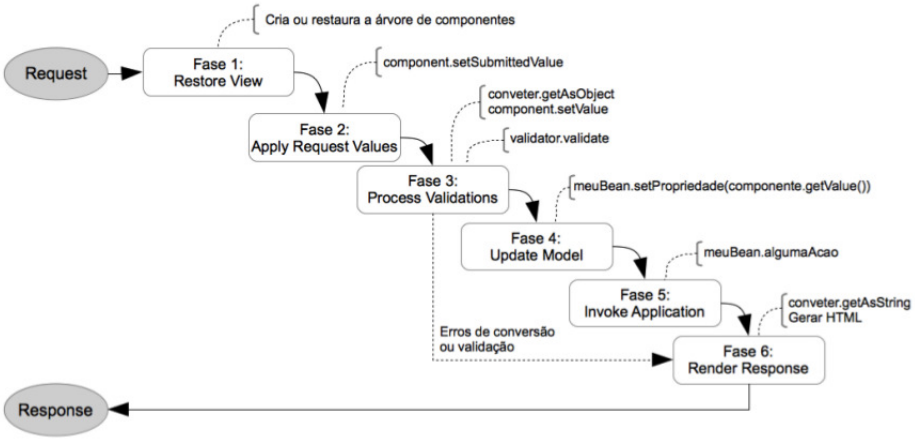

Figura 4 – Visão geral do ciclo de vida do JSF(CORDEIRO, 2013)

foi escolhido para o desenvolvimento. Ambos os dois necessitam que estes sejam inseridos no arquivo de dependências do *Maven*[6].

2.2.3 Hibernate

O Hibernate é um implementação da especificação de JPA, JavaPersistence API, que é um gerenciamento relacional das entidades do sistema com o banco de dados (JOHNSON, 2005). Esta ferramenta auxilia os desenvolvedores a ter uma conexão com o banco de dados bem parecido com as ideias/implementação do paradigma de programação, neste caso o Orientação a objetos (OO), onde ele permite, por exemplo, o uso de: polimorfismo, composição, herança.

O Hibernate não faz apenas o relacionamento de tabelas do dado com classes ou modelos em Java, ele provê uma abstração de funções que facilitam e pode significativamente reduzir o tempo gasto com o manuseio de *querys* em SQL ou a configuração de um JDBC (Java DataBase Communicator) (LIU; CHEN, 2009).

2.2.4 Arquillian

A ferramenta de software deve ser uma ferramenta que atenda de forma consistente, sem surpresas, a necessidade do usuário, por isso é necessário que haja bateria de testes a serem feitos durante o desenvolvimento de software. O processo de teste de software são as ações que se utilizam para evidenciar antes da implantação ou durante a fase de desenvolvimento, que o código produzido está fazendo o que deve ser feito corretamente (MYERS; SANDLER; BADGETT, 2011). Neste trabalho a ferramenta escolhida foi o

[6] Apache Maven é uma ferramenta de gerenciamento de projetos de software e compreensão. Baseado no conceito de um modelo de objeto do projeto (POM), Maven pode gerir construção de um projeto, elaboração de relatórios e documentação a partir de uma peça central de informações. Texto retirado do site: http://maven.apache.org/

Arquillian, que auxilia desenvolvedores a testar, de forma integrada, o uso dos *frameworks* JSF e testes unitários.

Segundo Ament (2013), o Arquillian é uma ferramenta de teste para Java que através dos *framework* de testes unitários, JUnit ou TestNG, executam casos de teste contra um container Java. A estrutura do Arquillian utiliza uma dependência externa, que define a implantação e carrega as classes necessárias para executar o teste. Outro benefício do uso do Arquillian é a possibilidade da integração do CDI para executar teste de diferentes partes do sistema com a utilização de injeção de dependência,tendo como base o testes unitários.

Os testes unitários são testes feitos para testar partes pequenas do sistema, quer seja um objeto de alguma classe com seus atributos, quer seja um método dentro do sistema, e os teste de integração são casos de testes onde se tenta testar partes acopladas do sistema (AMENT, 2013). Utilizando esses 2(dois) tipos de testes, o Arquillian consegue evidenciar que a codificação foi desenvolvida corretamente com os requisitos levantados.

Na próxima etapa será exibido como o sistema foi implementado, respeitando as decisões da arquitetura.

2.3 Etapa 3

Conforme falado no 1.5.3, esta etapa é a fase do trabalho onde é realizado o desenvolvimento da aplicação, esse desenvolvimento contempla a construção de um protótipo de ferramenta e a confecção de testes unitários para testar se o funcionamento do sistema está correto. Esta ferramenta serviu para automatiza o processo de avaliação do modelo BSIMM, é considerável falar mais uma vez que esta avaliação não segue nenhum documento formal (como por exemplo a ISO/IEC 15504), pode-se dizer que a ferramenta auxiliou na automatização de um fluxo de avaliação, tomando o SCAMPI como exemplo. O fluxo considerados assistidos pela ferramenta são os fluxos de condução da avaliação e resultados, vale ressalta que a ferramenta não substitui uma avaliação formal, porém esta auxilia as empresas ou auditores a executa uma avaliação de uma forma mais rápida.

O desenvolvimento da aplicação deu-se inicialmente pelos requisitos conhecidos como CRUD (*Create*, *Recover*, *Update* and *Delete* ou Inserir, recuperar, atualizar e deletar), esses requisitos foram definidos com base na dependência que as funcionalidades apresentavam. Através das decisões tomadas com base em experiência de implementação do desenvolvedor e a concepção de arquitetura inicial de arquitetura apresentado na figura 3, através disso foi gerado numa ideia de diagrama de pacotes e as relações entre si, conforme a Fig. (5).

Nessa figura é possível ver a distribuição de responsabilidades entre os pacotes do

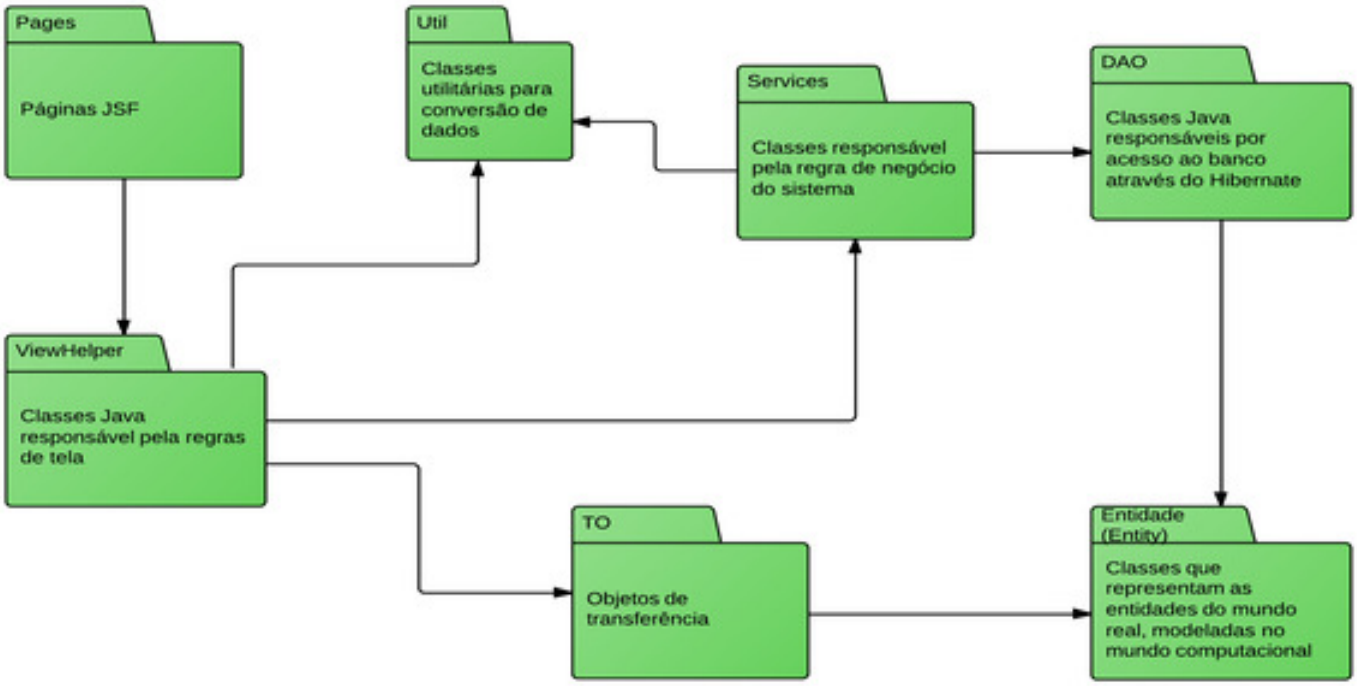

Figura 5 – Diagrama de pacotes do sistema

sistema, com isso tem uma coesão nas classes, porém é visível que há 2 (dois) pacotes que chamam atenção, que no caso são o *ViewHelper* e o TO. O pacote *ViewHelper* e *TO* são padrões de desenvolvimento que auxiliam o desenvolvedor a ter uma acoplamento e coesão correto além de aumentar o nível de manutenção sobre estes componentes.

As classes pertencentes ao pacote *TO - Transfer Object* tem como maior propósito reduzir o número de requisição direto com as classes do modelo, ou as *entity*. Um objeto de transferência é criado no servidor e é responsável por inserir os valores nas classes modelos e retornar apenas o objeto, sem a necessidade de conhecimento dos atributos das classes, eles tem um maior interação com as classes do pacote *ViewHelper*. (INDUKURI; ASTHANA,). O padrão *ViewHelper* é criado para encapsular a lógica de tela e delegar a lógica de negócio a classes subjacentes, mais especificamente as classes do pacote *Service*, com esse padrão é possível uma distinção entre as diferentes camadas do sistema (HAMMOUDA; KOSKIMIES, 2002).

Outro padrão de desenvolvimento notável no sistema é o padrão *Data Acess Object* ou DAO, onde são inseridas classes responsáveis por acesso ao banco de dados, fazendo assim que as camadas superiores não tenham qualquer conhecimento do banco de dados da aplicação, outro benefício é a de prover uma facilidade na transferência de ambiente da aplicação (trocar o tipo de banco de dados, por exemplo) (INDUKURI; ASTHANA,).

O desenvolvimento da aplicação foi feito seguindo o modelo de dados/classe descrito na Fig. (6). É possível visualizar os relacionamentos e com isso criando uma depen-

dência entre as funcionalidades do sistemas.

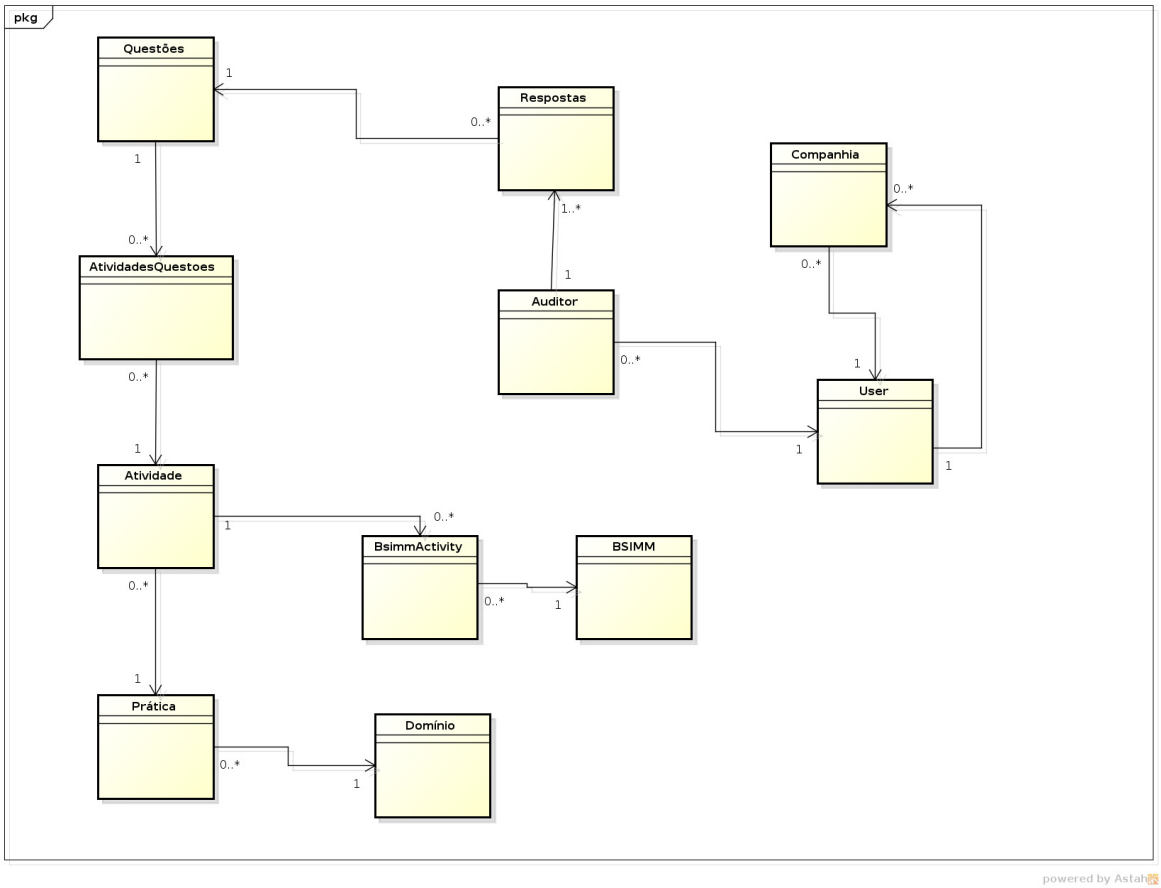

Figura 6 – Modelo de Dados do sistema

O desenvolvimento da funcionalidade de manter um companhia leva em consideração apenas o nome da companhia. Entretanto, fica notável que uma companhia tem um responsável para o seu cadastro. Este responsável é um usuário do sistema, este usuário é composto de um perfil, este perfil é responsável pelas atribuições dentro do sistema, onde-se tem o perfil de Usuário e Administrador.

Partindo para o desenvolvimento do modelo, tem-se a distribuição de suas partes. O modelo apenas altera a sua versão ao longo do tempo, não necessitando de mudança no nome, por isso a entidade do modelo tem apenas este atributo, para o usuário ter a possibilidade de cadastrar uma nova versão, sem que haja inserção direta na base de dados.

Conforme as Figuras (1) e (2) é possível visualizar que o modelo possui 3 partes a serem implementadas. Partindo de uma observação de maior número de relacionamento, um cadastro de atividade, é necessário que se tenha domínios e práticas na base de dados. Porém, após uma análise sobre as diferentes versões já lançadas do modelo, constatou-se que, ambos, domínios e práticas não se alteraram, por isso decidiu-se apenas deixá-los gravado na base de dados, sem a necessidade de um implementação no código para estas

2 (duas) entidades.

Entrando no modelo da auditoria, é possível visualizar que pode-se ter uma questão com vários relacionamentos com atividades. Isso se deve ao fato de uma questão ter a possibilidade de atender a diversas atividades, o que não é comum, porém a modelagem foi feita pensado em casos que esta situação poderiam acontecer. Após todas as etapas anteriores serem seguidas, a ordem importa, uma vez que não tem sentido cadastrar uma questão que não atender uma atividade, é passado para a fase de auditoria.

A auditoria de uma companhia dar-se-á pela possibilidade do usuário utilizar questões de acordo com o escopo da avaliação, ou seja, o usuário tem escolha livre em cadastrar uma nova auditoria partindo de selecionar uma, duas ou mais práticas, selecionar o nível que ele deseja verificar, além da versão do modelo.

A auditoria também tem uma peculiaridade, pois é possível utilizar questões de uma auditoria já cadastrada, isso serve para o auditor ter a possibilidade de cadastrar uma nova auditoria e se desejar, comparar com a já realizada, para uma análise do estado em relação ao tempo decorrido, por isso é necessário o atributo de data de uma auditoria, onde é possível buscar auditoria pelo dia que foi realizada.Para um melhor entendimento do fluxo a ser seguido no sistema, é apresentado na Fig. (7) o diagrama de sequência do sistema, uma versão inicial.

No primeiro momento a linguagem escolhida para o desenvolvimento foi o Ruby juntamente com o *framewok* Rails, o famoso Ruby on Rails. A linguagem Ruby foi criada por Yukihiro Matsumoto em 1995 no Japão, seguindo a OO. Além das características desse paradigma, o ruby foi criado com o intuito de ser uma linguagem funcional, acrescendo recursos que não constavam no OO (SOUZA, 2013).O Rails é um *framework* de uso livre para desenvolvimento web, desenvolvido em 2003, que tem por base a arquitetura *Model-View-Controller* (MVC) que é uma arquitetura que tem a divisão de responsabilidades entre elas, a camada *model* é a camada responsável pelas entidades do mundo real,a camada *view* é responsável pela apresentação na tela do usário e a camada *controller* é a camada intermediária, ela que faz a transformação da camada de modelo para a de apresentação (FUENTES, 2013).

A linguagem Ruby ofereceu em certo momento uma velocidade de desenvolvimento. Entretanto, em um certo momento do desenvolvimento, a continuação do projeto foi colocado em risco, uma funcionalidade do sistema ficou parada mais de 3 (três) semanas e a falta de experiência do desenvolvedor em relação a Ruby on Rails, o sistema foi migrado de um aplicação Ruby para uma aplicação Java WEB.

A seguir são apresentadas as imagens do sistemas já em execução, essas imagens são exemplos da execução do sistema, contudo é a execução do sistema apenas tendo como base uma prática do modelo. Essa prática é a mais conhecida pelo usuário, no qual pediu

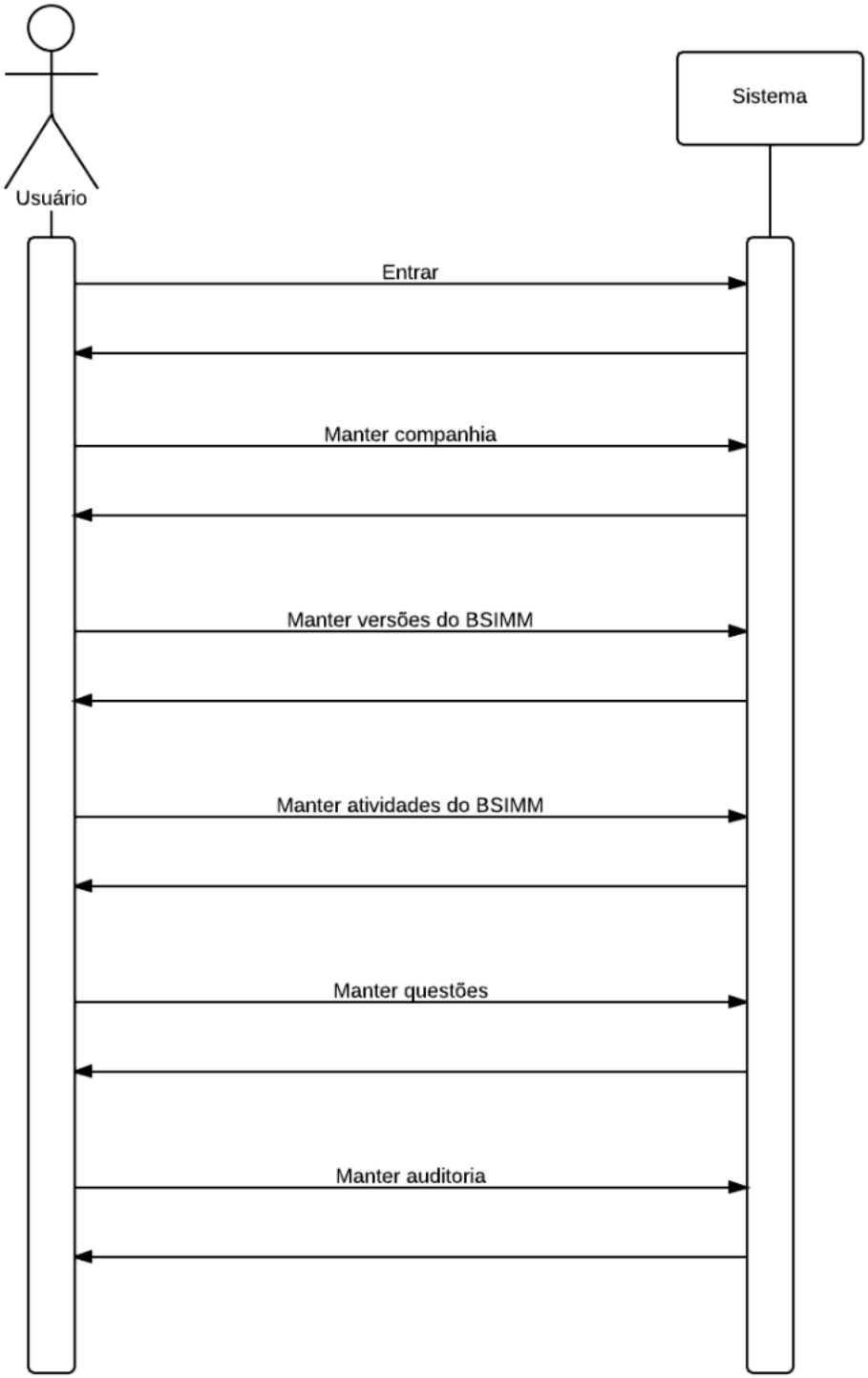

Figura 7 – Diagrama de sequência do sistema

para não ser identificado, pois é uma empresa de tecnologia de Brasília que envolve uma gama de usuário e aplicações que seriam necessários práticas de segurança em todo o seu desenvolvimento.

A figura 2.3 mostra a página inicial do sistema, onde todas as senhas são criptografadas com a utilização do algoritmo SHA-256, no qual é um algoritmo para criptografia

baseada na função HASH. Após a etapa de login, o sistema verifica as permissões do usuário e assim mostra a sua estrutura de tela.

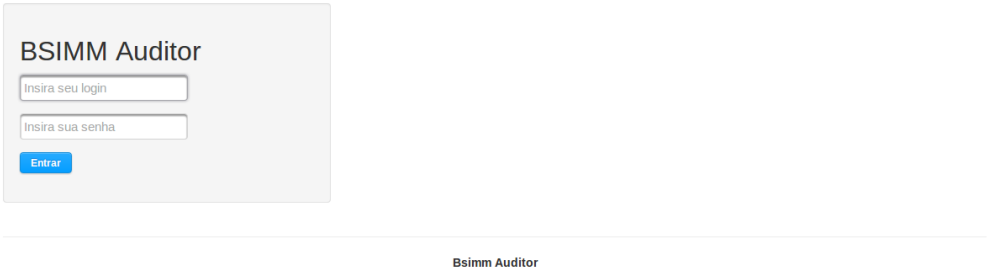

Figura 8 – Página de login

A figura 2.3 mostra já mostra a página inicial do perfil logado, esse perfil é o de usuário, onde o usuário tem a visualização das ações que pode fazer pelo menu. É importante frisar que as modificações em relação ao modelo, como cadastro de práticas e atividades, são de responsabilidade de usuários com perfis de administrador, pois são cadastros que devem ser espelho do modelo e com isso não haver muitas alterações.

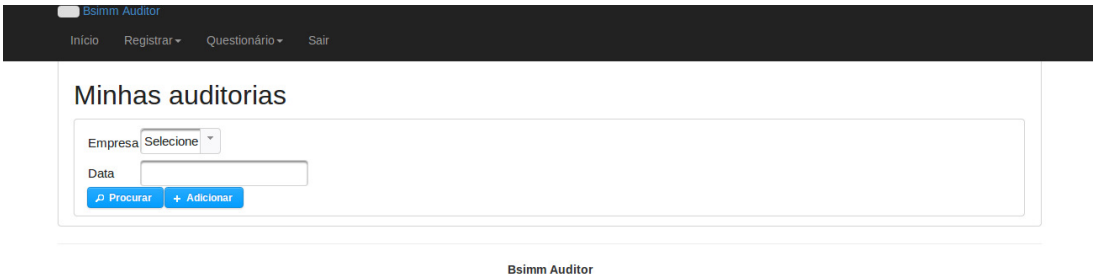

Figura 9 – Página de visualização das auditorias do perfil logado

A figura 2.3 mostra a ação do usuário em selecionar as empresas de acordo com seu perfil, ou seja, somente empresas cadastrados pelo usuário ou empresas que foi vinculado ao usuário corrente é mostrado, essa visualização é uma forma de controle para saber

qual usuário é responsável pela empresa ou qual empresa o auditor participa. A figura 2.3 mostra o resultado das pesquisas da auditoria pela busca de data, essa data é a data que a auditoria foi salva no sistema, essa auditoria foi criada e salva, se for editada a data continua sendo a mesma, pois foi criada num momento anterior.

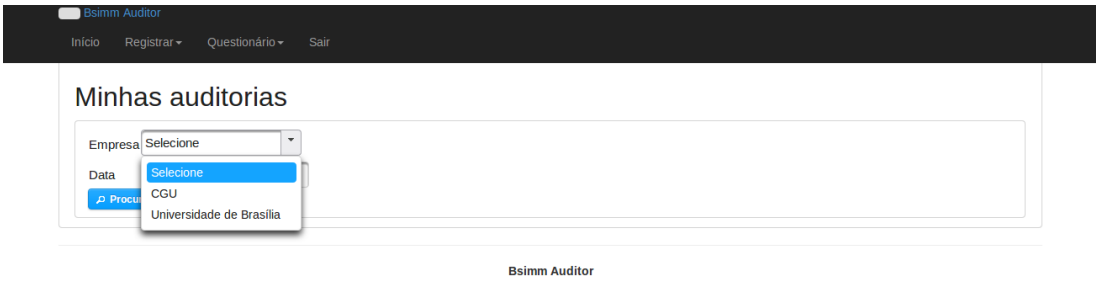

Figura 10 – Tela de seleção das empresas vinculadas ao perfil

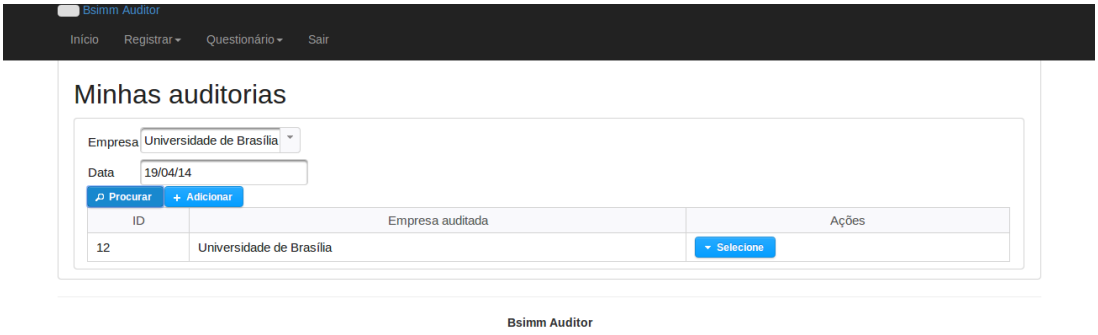

Figura 11 – Página dos resultado da busca de auditorias pela data

A figura 2.3 mostra a página de criação de uma nova auditoria. Nesta página consta os parâmetros para a criação de uma nova auditoria, seguindo o informado pelo usuário. Caso os parâmetros (nível, práticas e versão do modelo) informados sejam iguais a uma auditoria já cadastrada e que esteja encerrada, o sistema irá apresentar um diálogo para saber se o usuário deseja reutilizar a auditoria, conforme fig. 2.3, isso é, utilizar as questões para responder uma nova auditoria.

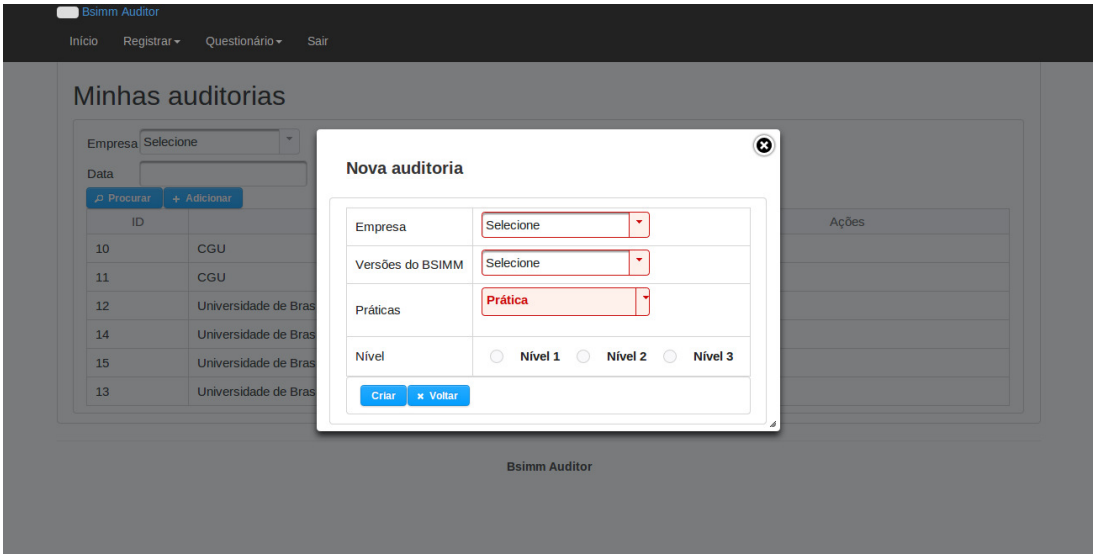

Figura 12 – Página para criar uma nova auditoria

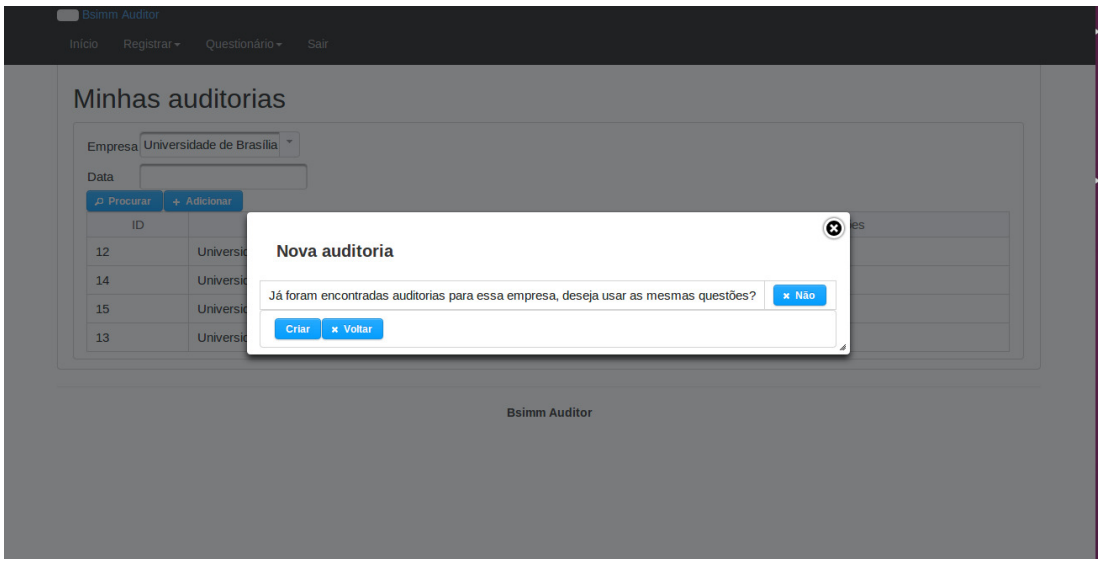

Figura 13 – Diálogo que aparece caso uma opção de auditoria exista, com os mesmo parâmetros de uma nova

A figura 2.3 mostra como é a tela de uma auditoria, onde se tem as questões buscadas de acordo com os parâmetros informados. Essa auditoria apenas mostra as questões para o usuário responder, essa resposta é uma resposta binário, ou sim ou não, conforme visto na fig. 2.3.

A figura 2.3 mostra o gráfico de teia criado após a auditoria ter sido criada e respondida, não sendo necessário a finalização da auditoria para a geração do gráfico. Esse gráfico é foi criado com base em gráficos que tem no modelo onde é citado as informações de todas as empresas cadastradas na pesquisa.

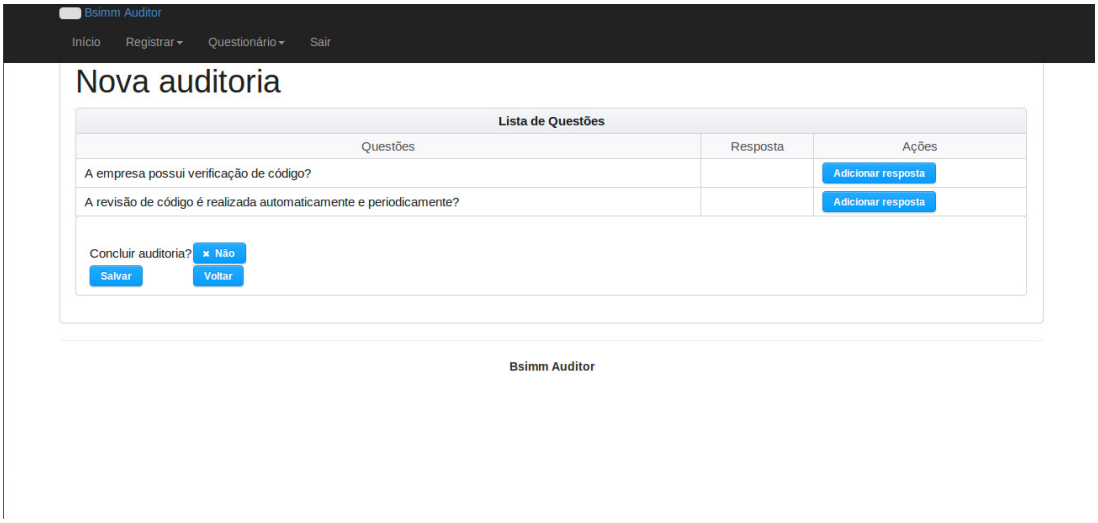

Figura 14 – Página de uma nova auditoria

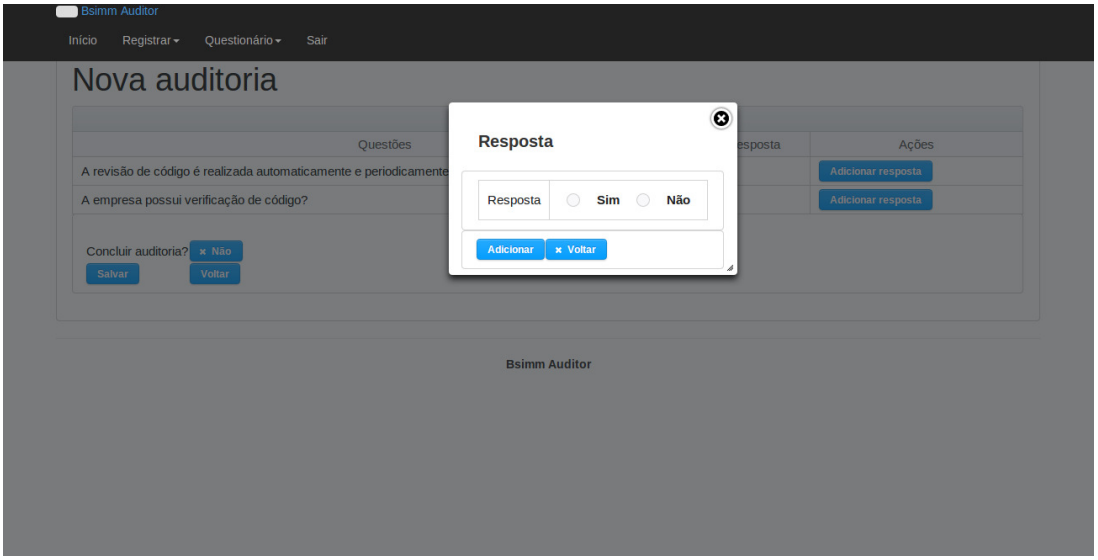

Figura 15 – Diálogo para inserir uma nova resposta

2.3.1 Testes

O desenvolvimento dos testes foi também efetuado com base no diagrama de sequência, pelo menos a questão do fluxo de desenvolvimento. Os testes foram efetuados com intuito de verificar que a ferramenta está com o funcionamento correto, isso é, os testes verificam que as funções básicas do sistema, que são as funções *CRUD*, conforme citado na seção 2.3. Além de testar se a injeção de dependência funciona de forma correta e estará funcionando quando o usuário estiver acessando a funcionalidade através de tela.

O desenvolvimento dos testes seguiu um princípio quando se implementa uma bateria de testes unitários, ou seja, um teste não deve impactar no outro e com isso deve haver meios de sanar a dependência entre as classes. Essa dependência, ou necessidade de ser ter registros previamente cadastrados na base de dados é sanado com um método

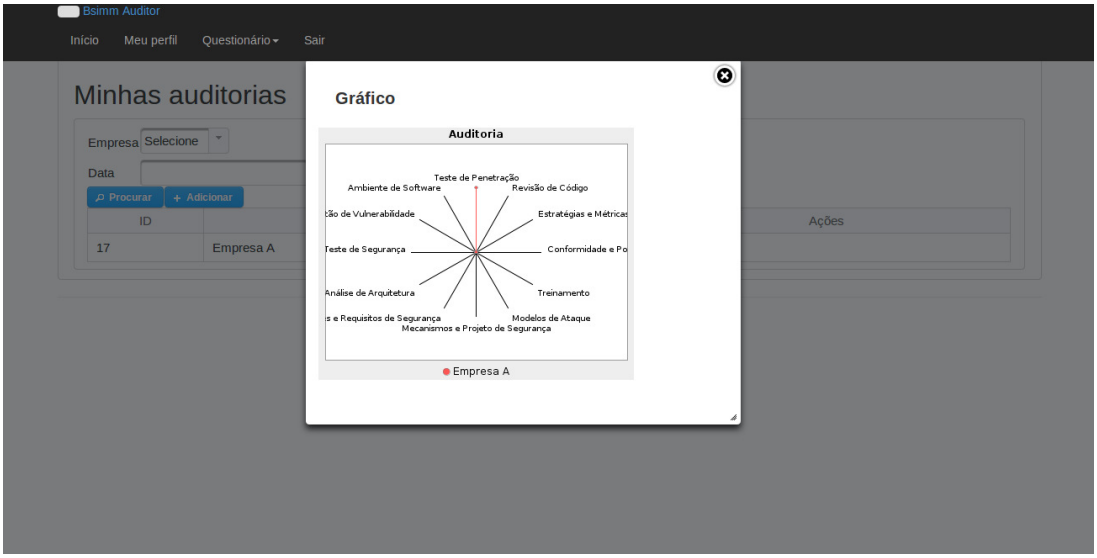

Figura 16 – Gráfico de teia sobre a auditoria somente em uma prática

chamado de *Before*, ou antes. Este método é executado antes da bateria de teste e é responsável por inserir uma base de dados para que os testes possam ser realizados da melhor forma possível.

A confecção dos testes deu-se pela invocação dos métodos criados para executar funções dentro da aplicação, isso é, os testes são realizados para inserção de dados, com os métodos que serão usado pelo usuário. Isso se deve ao fato do teste executar uma ação que será executado pelo usuário.

Os testes de integração serviu para testar se houve a necessidade de exigir uma classe a mais para o desenvolvimento da funcionalidade. Os testes de integração testam as partes do sistemas de forma acoplada. Dentro dos testes são colocadas os objetos das classes necessárias, isso é, além de colocar dentro do arquivo de configuração do Arquillian, para cada caso de teste, é necessário injetar um objeto dentro do contexto de funcionalidade e assim chamar os métodos que esta possui.

2.4 Exemplo de Uso

Para exemplificar o uso da ferramenta, foi realizado uma auditoria, fictícia, com um contexto de uso da ferramenta durante um período. No início, a auditoria tinha o objetivo de mostrar a todos como se encontrava a empresa antes da adoção do BSIMM no seu processo. A auditoria envolveu um questionário com todas as práticas em todos os níveis. O resultado dessa auditoria pode ser visto na fig. 2.4.

Pode-se verificar que a empresa apresenta um grau baixo de aderência em relação ao modelo, isso deve pelo processo de segurança não acontecer da forma mais correta, em muitos casos, segurança de software é confundido como técnicas e/ou ferramentas de

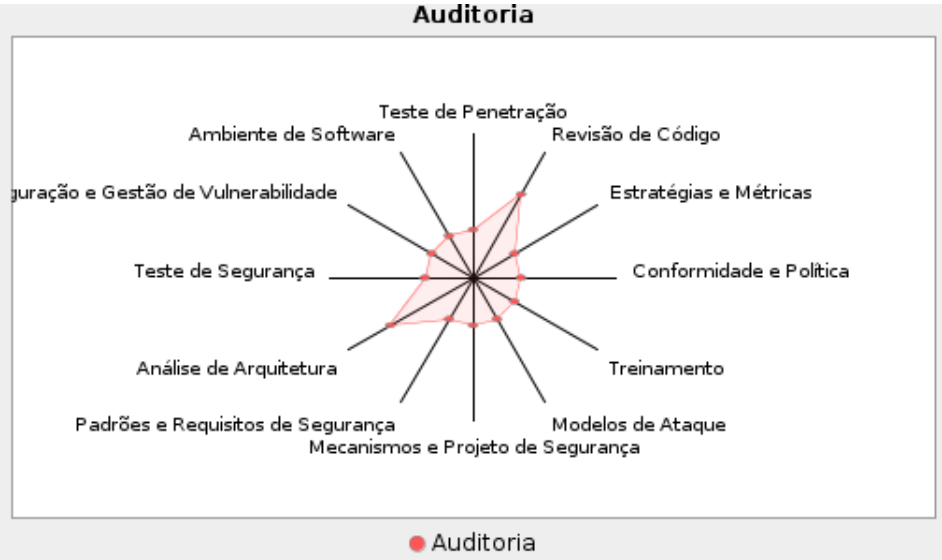

Figura 17 – Gráfico de teia sobre a auditoria

seguranças. Por isso em alguns casos, como exemplo o caso da prática de Revisão de código onde apresenta o nível 2, pois uma atividade executada antes da adesão já era executada.

Em segundo momento de tempo, com o insumo do diagnóstico inicial, foi determinado que a empresa tinha como alvo o nível 2 em todas as práticas avaliadas, e foram escolhidas apenas as práticas de Gestão de Vulnerabilidades e Gestão de Configuração, Análise de Arquitetura, Estratégias e Métricas, além de manter o nível na prática que já se encontrava. Nessa etapa a empresa apresentou mais maturidade no processo de segurança, uma vez que tendo conhecimento e possuindo um guia houve facilidade em adoção de novas atividades. O resultado da auditoria pode ser visto na fig. 2.4.

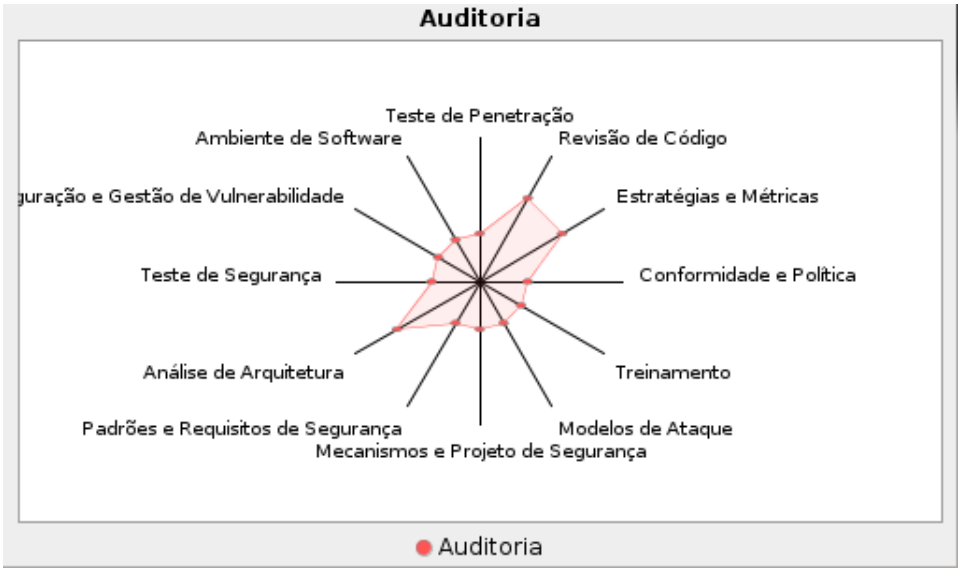

Figura 18 – Auditoria em um segundo momento

A terceira etapa da realização é o passo antes de ser feito uma auditoria real. Esta auditoria é encarada como uma teste para a auditoria e o objetivo desta será o nível 2 para todas as práticas, e se possível nas práticas que já se encontravam neste patamar subir o nível. O objetivo desta auditoria também é identificar pontos que podem ser melhorados no último momento e com isso auxiliar no resultado.

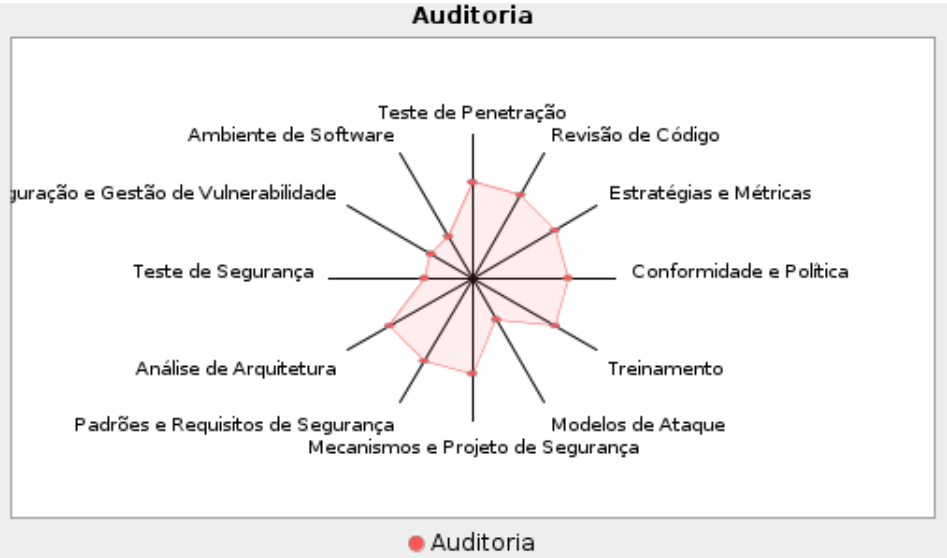

Figura 19 – Terceira auditoria

3 Considerações Finais

Conforme citado na seção 1.3, este trabalho possui etapas a serem seguidas para o seu desenvolvimento. Os objetivos específicos foram atendidos, uma vez, que o objetivo do trabalho é a de prototipar uma ferramenta, ou seja, não é ter uma ferramenta com todas as funcionalidades desejadas, porém ter um esqueleto ou uma ideia previamente implementada em mundo computacional.

Para a construção do trabalho foi necessário levantar os modelos de maturidade pautadas no desenvolvimento de software seguro durante todo o *Software Development Life Cycle* (SDLC). Ambos os modelos foram estudados, e a escolha do modelo deu-se pela sua filosofia, de ser uma prescrição de atividades que são executadas em empresas que já adotam a construção de software seguro em seu desenvolvimento.

O desenvolvimento da ferramenta levou em conta a experiência e o conhecimento que o desenvolvedor possuía, por isso as ferramentas escolhidas, citadas na seção 2.2, foram somente voltadas para o desenvolvimento *WEB* e relacionada a linguagem Java. Essa talvez não possa ter sido as melhores escolhas para o desenvolvimento, porém atendeu bem ao desenvolvimento, com algumas ressalvas.

A codificação foi a etapa que teve uma maior duração no desenvolvimento do trabalho. A modelagem de dados deu a ideia inicial de como ficaria o relacionamento entre as partes da ferramenta, e como seria o fluxo para um bom desenvolvimento da ferramenta. Entretanto, viu-se que a codificação com uma linguagem que não se tem experiência pode deixar o trabalho mais árduo e com isso atrapalhar o desenvolvimento. A mudança, como sempre em software, foi vista com bons olhos e necessário para o término da ferramenta.

Dentro da implementação, a etapa de visualização é a de exibição do gráfico. Esse gráfico é uma implementação com a biblioteca *free*, o JfreeChart. Porém essa biblioteca tem uma limitação no seu uso, a exibição dentro da uma página JSF não é considerada boa, pois para exibir é transformado o gráfico em uma imagem. A exibição do gráfico também é um ponto fraco do uso desta biblioteca, nesta biblioteca a legenda apresenta apenas uma informação e o gráfico também não exibe os pontos (informações compostas por numerais por exemplo) para o usuário.

Os testes foram feitos apenas para a verificação das funcionalidades de forma inicial, isso é, as funcionalidades de manter as entidades dentro da base de dados para este ser utilizado na auditoria. O sucesso dos testes deu-se especialmente pelo uso da ferramenta Arquillian.

3.1 Trabalhos Futuros

Futuramente, em uma possibilidade de *pull request*, ou seja, adicionar uma nova funcionalidade a ferramenta, seria a de dar a possibilidade ao auditor de inserir uma comprovação da resposta de uma questão. Ou seja, seria possível em cada resposta, seja ela sim ou não, o carregamento de um arquivo (documento, foto, entrevista em áudio, vídeo) comprobatório daquela resposta.

Outra opção seria a mudança na forma de visualizar o gráfico. Atualmente na ferramenta, a visualização do gráfico é feita com o uso da biblioteca *free* Jfreechart. Porém, essa biblioteca tem uma deficiência na visualização, e sua comunicação com o *JSF* não é considerada uma comunicação facilitada, porém alterando a visualização do gráfico para o uso da API, Highchart, seria uma forma de deixar o gráfico mais dinâmico e mais fácil de ser entedido.

Referências

AMENT, J. *Arquillian Testing Guide*. Birmingham, UK: Packt Publishing Ltd., 2013. 227 p. ISBN 9781782160700. Disponível em: <www.packtpub.com>. Citado 2 vezes nas páginas 32 e 33.

ANACLETO, A.; WANGENHEIM, C. *Método e modelo de avaliação para melhoria de processos de software em micro e pequenas empresas*. Tese (Doutorado) — Universidade Federal de Santa Cantarina, 2004. Disponível em: <http://www.gqs.ufsc.br/wp-content/uploads/2011/11/Dissertacao_AlessandraAnacleto.pdf>. Citado na página 25.

BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. The goal question metric approach. *Encyclopedia of software engineering*, v. 2, p. 1–10, 1994. Disponível em: <https://xa-.yimg.com/kq/groups/19378749/392126381/name/GQM-paper.pdf>. Citado na página 25.

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice*. Addison Wesley Professional, 2003. (The SEI Series in Software Engineering). ISBN 9780321154958. Disponível em: <http://books.google.com.br-/books?id=mdiIu8Kk1WMC>. Citado na página 27.

BERGSTEN, H. *JavaServer Faces*. O'Reilly Media, 2004. ISBN 9781449378950. Disponível em: <http://books.google.com.br/books?id=oZVTuH67oWgC>. Citado na página 30.

BORANBAYEV, A. S. Defining methodologies for developing J2EE web-based information systems. *Nonlinear Analysis: Theory, Methods & Applications*, Elsevier Ltd, v. 71, n. 12, p. e1633–e1637, dez. 2009. ISSN 0362546X. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S0362546X09002570>. Citado na página 29.

CHESS, B.; ARKIN, B. Software Security in Practice. *IEEE Security & Privacy Magazine*, v. 9, n. 2, p. 89–92, mar. 2011. ISSN 1540-7993. Disponível em: <http:-//ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5739646>. Citado na página 18.

CHRISSIS, M. B.; KONRAD, M.; SHRUM, S. *CMMI Guidlines for Process Integration and Product Improvement*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 0321154967. Citado na página 21.

CORDEIRO, G. S. *CDI: Integre as dependências e contextos do seu código Java*. São Paulo: Casa do Código, 2013. 216 p. ISBN 978-85-66250-18-3. Citado 3 vezes nas páginas 11, 29 e 31.

FUENTES, V. *Ruby on Rails: Coloque sua aplicação web nos trilhos*. Casa do Código, 2013. ISBN 9788566250039. Disponível em: <http://books.google.com.br-/books?id=bfc7nQEACAAJ>. Citado na página 36.

GUSTAFSSON, J. et al. Architecture-centric software evolution by software metrics and design patterns. *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, IEEE Comput. Soc, p. 108–115, 2002. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=995795>. Citado na página 27.

HAMMOUDA, I.; KOSKIMIES, K. A pattern-based J2EE application development environment. *Nord. J. Comput.*, 2002. Disponível em: <http://practise2.cs.tut.fi/pub-/papers/NWPER02_Imed.pdf>. Citado na página 34.

INDUKURI, R.; ASTHANA, R. Design Patterns Framework and Pattern Oriented Design Process. *innovateapps.com*. Disponível em: <http://www.innovateapps.com/pdfs-/3_Design_Patterns_Framework_and_Pattern-Oriented_Design_Process.pdf>. Citado na página 34.

ITABORAHY, A. et al. *Aplicação do método SCAMPI para avaliação do processo de gerenciamento de projetos de software numa instituição financeira*. 2005. Disponível em: <http://www.simpros.com.br/upload/A02_1_artigo14748.pdf>. Citado na página 26.

JOHNSON, R. J2EE Development Frameworks. n. January, p. 107–110, 2005. Citado na página 32.

KING, G. *JSR-299: Contexts and Dependency Injection for the Java EE platform*. [S.l.], 2009. Citado na página 30.

LIU, S.; CHEN, P. Developing Java EE Applications Based on Utilizing Design Patterns. *2009 WASE International Conference on Information Engineering*, Ieee, p. 398–401, jul. 2009. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper-.htm?arnumber=5211375>. Citado na página 32.

MCGRAW, G. Software Security. *Datenschutz und Datensicherheit - DuD*, v. 36, n. 9, p. 662–665, set. 2012. ISSN 1614-0702. Disponível em: <http://link.springer.com/10.1007-/s11623-012-0222-3>. Citado na página 21.

MCGRAW, G.; MIGUES, S.; WEST, J. *Building Security In Maturity Model*. [S.l.], 2013. Disponível em: <http://w.secappdev.org/handouts/2010/Gary McGraw/bsimm 15things lo res.pdf>. Citado 5 vezes nas páginas 11, 22, 23, 24 e 26.

MYERS, G.; SANDLER, C.; BADGETT, T. *The Art of Software Testing*. Wiley, 2011. (ITPro collection). ISBN 9781118133156. Disponível em: <http://books.google.com.br-/books?id=GjyEFPkMCwcC>. Citado na página 32.

PRESSMAN, R. *Engenharia de software*. McGraw-Hill, 2006. ISBN 9788586804571. Disponível em: <http://books.google.com.br/books?id=MNM6AgAACAAJ>. Citado na página 27.

PRIKLADNICKI, R.; BECKER, C.; YAMAGUTI, M. *Uma Abordagem para a Realização de Diagnóstico Inicial em Empresas que Implementam o MPS. BR*. 2005. Disponível em: <http://www.softex.br/wp-content/uploads/2013/09-/MPSBR2005_Artigo_Softsul_WSI_7ago05.pdf>. Citado na página 26.

RAZINA, E. et al. Effects of dependency injection on maintainability. p. 7–12, 2007. Citado na página 28.

REGULWAR, G. B.; GULHANE, V.; JAWANDHIYA, P. M. A Security Engineering Capability Maturity Model. *2010 International Conference on Educational and Information Technology*, Ieee, n. Iceit, p. 306–311, set. 2010. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5607700>. Citado na página 21.

SOUZA, L. *Ruby: Aprenda a programar na linguagem mais divertida*. [S.l.]: Casa do Código, 2013. ISBN 9788566250039. Citado na página 36.

TONINI, A. C.; CARVALHO, M. M. D.; SPINOLA, M. D. M. Contribuição dos modelos de qualidade e maturidade na melhoria dos processos de. p. 275–286, 2008. Citado na página 21.

VIEGA, J. Ten Years of Trustworthy Computing: Lessons Learned. *IEEE Security & Privacy Magazine*, v. 9, n. 5, p. 3–4, set. 2011. ISSN 1540-7993. Citado na página 18.

WILLIAMS, B. J.; CARVER, J. C. Characterizing software architecture changes: A systematic review. *Information and Software Technology*, Elsevier B.V., v. 52, n. 1, p. 31–51, jan. 2010. ISSN 09505849. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S0950584909001207>. Citado na página 27.